Programación híbrida en clusters de multicore. Análisis del impacto de la jerarquía de memoria.

Fabiana Leibovich[1], Franco Chichizola[1], Laura De Giusti[1], Marcelo Naiouf[1], Francisco Tirado Fernández[2], Armando De Giusti[1 3]

[1]Instituto de Investigación en Informática LIDI (III-LIDI), Facultad de Informática, Universidad Nacional de La Plata, 50 y 120 2do piso, La Plata, Argentina.
{fleibovich, francoch, ldgiusti, mnaiouf, degiusti}@lidi.info.unlp.edu.ar
[2]Departamento de Arquitectura de Computadores y Automática, Facultad de Informática, Universidad Complutense de Madrid, Madrid, España.
{ptirado}@dacya.ucm.es
[3] CONICET

Abstract. Desde la aparición de las arquitecturas *clusters* de *multicore*, la programación híbrida surgió como una herramienta importante para el aprovechamiento de la nueva jerarquía de memoria que la arquitectura impone. Partiendo de un mismo caso de estudio, este trabajo se enfoca en la comparación de dos soluciones híbridas (donde se combina pasaje de mensajes y memoria compartida) que resuelven el problema en cuestión utilizando diferentes estrategias de paralelización. Las mismas utilizan de distinta manera la jerarquía de memoria presente en la arquitectura de experimentación (cluster de *multicore*), en particular haciendo un uso diferente del nivel L1 de *cache*. El caso de estudio elegido es el problema clásico de multiplicación de matrices, utilizado para demostrar el impacto de la utilización óptima de la jerarquía de memoria existente en una arquitectura paralela.

Keywords: arquitecturas paralelas, programación híbrida, *cluster*, *multicore*, jerarquía de memoria, rendimiento.

1 Introducción

La última gran evolución en las arquitecturas paralelas son los *clusters* de *multicore* [1]. El mismo consiste en un conjunto de procesadores de múltiples núcleos interconectados mediante una red, en la que trabajan cooperativamente como un único recurso de cómputo. Es similar a un *cluster* tradicional pero con la diferencia de que cada nodo posee al menos un procesador *multicore* en lugar de un monoprocesador. Esto permite combinar las características más distintivas de ambas arquitecturas (memoria distribuida y compartida), dando origen a los sistemas híbridos [2].

Al diseñar un algoritmo paralelo es muy importante considerar la jerarquía de memoria con la que se cuenta, ya que es uno de los factores que incidirá directamente en la performance alcanzable del mismo. Los *clusters* de *multicore* introducen un nivel más en la jerarquía de memoria si se lo compara con los *multicore*: la memoria

distribuida accesible vía red; que permite la interconexión de los diferentes procesadores que conforman el *cluster*. Si se enumera la jerarquía de memoria, la misma queda conformada de la siguiente manera: niveles de registros y caché L1 propio de cada núcleo, *cache* compartida de a pares de núcleos (L2), memoria compartida entre los *cores* de un procesador *multicore* y finalmente memoria distribuida vía red [3], tal como puede verse en la Figura 1.

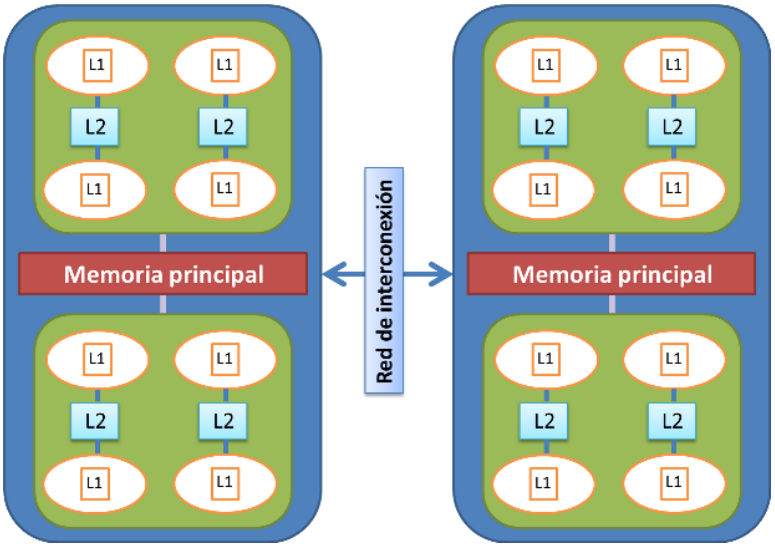

Fig. 1. *Cluster* de *multicore*

Es necesario aclarar que esta jerarquía puede ampliarse según los modelos de procesadores utilizados. Actualmente es frecuente encontrar procesadores *multicore* que utilizan un tercer nivel de *cache* (L3) [4].

En este trabajo se ha seleccionado como caso de estudio la multiplicación de matrices, una de las aplicaciones más tradicionales y estudiada en el cómputo paralelo (dado que constituye la base de otras). Los motivos principales por los cuales se utiliza esta aplicación (ampliamente probada y evaluada por los autores [5][6] y los profesionales especializados en el área de interés de este trabajo) son que la misma permite utilizar y explotar el paralelismo de datos, así como la facilidad que presenta para escalar el problema al aumentar el tamaño de las matrices [7]. En esta aplicación los problemas de sincronización son mínimos y permiten enfocarse directamente en el aprovechamiento de la jerarquía de memoria, objetivo principal de esta investigación. De esta forma, pueden compararse soluciones híbridas que aprovechan de manera diferente la jerarquía de memoria disponible en la arquitectura utilizada en la experimentación. Este articulo es una evolución de investigaciones previas, en las que se utilizó este mismo caso de estudio pero comparando la programación híbrida con la de pasaje de mensajes con diferentes estrategias de implementación [8].

El artículo está organizado de la siguiente manera: en la Sección 2 se detallan los aportes del trabajo, mientras que la Sección 3 describe los basamentos teóricos de la

jerarquía de memoria y las características de la programación híbrida. En la Sección 4 se detalla el caso de estudio, mientras que en la Sección 5 se muestran las soluciones implementadas. La arquitectura utilizada y los resultados se muestran en la Sección 6. Por último, en la Sección 7 se exponen las conclusiones y líneas de investigación futuras.

2 Aportes del trabajo

El principal aporte de este trabajo es contrastar dos soluciones híbridas que si bien resuelven el mismo problema, hacen un aprovechamiento diferente de la jerarquía de memoria subyacente, particularmente del nivel L1 de cache, haciendo un uso eficiente de la localidad espacial y temporal de los datos.

El análisis se realiza en base al tiempo de ejecución y eficiencia de las soluciones híbridas al escalar el tamaño del problema y la cantidad de núcleos utilizados.

3 Jerarquía de memoria y programación híbrida

A continuación se describen los conceptos teóricos sobre los que se basa esta investigación. Primero se presentan dos conceptos fundamentales relacionados a la jerarquía de memoria: localidad espacial y temporal de la cache. Luego se describe la programación híbrida como método de aprovechamiento de la arquitectura.

3.1 Localidad temporal y espacial de la cache

El concepto de localidad temporal hace referencia a que la cache mantiene los datos recientemente accedidos. Es decir que si se aprovecha el acceso a los datos teniendo en cuenta este factor, la latencia en el acceso a los mismos disminuirá. Por otro lado, el concepto de localidad espacial hace referencia a que cada vez que se trae un dato a cache, se obtiene una línea de la misma; de esta forma, si los datos accedidos son contiguos, también se disminuye la latencia en el acceso a los datos (en una sola lectura a memoria, los datos contiguos estarán ya en memoria *cache*) [9].

3.2 Programación híbrida

La programación híbrida combina la memoria compartida con el pasaje de mensajes [2][9], aprovechando sus potencialidades. La comunicación entre procesos que pertenecen al mismo procesador físico puede realizarse utilizando memoria compartida (nivel micro), mientras que la comunicación entre procesadores físicos (nivel macro) se suele realizar por medio de pasaje de mensajes (memoria distribuida).

En el primer caso, los datos accedidos por la aplicación se encuentran en una memoria global accesible por los procesadores paralelos. Esto significa que cada procesador puede buscar y almacenar datos de cualquier posición de memoria

independientemente uno de otro. Se caracteriza por la necesidad de la sincronización para preservar la integridad de las estructuras de datos compartidas.

En el pasaje de mensajes, los datos son vistos como asociados a un proceso particular. De esta manera, se necesita de la comunicación por mensajes entre ellos para acceder a un dato remoto. En este modelo, las primitivas de envío y recepción son las encargadas de manejar la sincronización.

El objetivo de utilizar el modelo híbrido es aprovechar y aplicar las potencialidades de cada una de las estrategias que el mismo brinda, de acuerdo a la necesidad de la aplicación. Esta es un área de investigación de interés actual, y entre los lenguajes que se utilizan para programación híbrida aparecen *OpenMP* [10] para memoria compartida y *MPI* [11] para pasaje de mensajes.

4 Caso de estudio

Dadas dos matrices A de *m x p* y B de *p x n* elementos, la multiplicación de ambas consiste en obtener la matriz C de *m x n* elementos (C = A x B), donde cada elemento se calcula por medio de la Ecuación 1.

$$C_{i,j} = \sum_{k=1}^{p} A_{i,k} * B_{k,j} \qquad (1)$$

5 Soluciones Implementadas

Los estudios experimentales se realizaron, por un lado, implementando el algoritmo de multiplicación de matrices en su versión secuencial. Para las soluciones paralelas, se implementaron dos variantes. En una de ellas los resultados (matriz C) se calculan por bloques, mientras que en el segundo algoritmo paralelo, el cálculo de la matriz C se realiza subdiviendo las matrices A y B en bloques para ser procesados. En ambas soluciones se utilizó el modelo de programación híbrida.

Tanto la solución secuencial como las paralelas fueron desarrolladas utilizando el lenguaje C. Las soluciones paralelas utilizan para el pasaje de mensajes la librería *OpenMPI* [11], mientras que para memoria compartida emplean *OpenMP* [10].

En este trabajo se realiza un análisis experimental del comportamiento de una aplicación híbrida y el aprovechamiento de la jerarquía de memoria subyacente en una arquitectura *cluster* de *multicores* [12][13][14].

Los resultados que se muestran se enfocan a analizar las soluciones híbridas en dos sentidos:

1. El comportamiento al incrementar el tamaño del problema y la cantidad de núcleos (escalabilidad) [15][16]. En este caso, se procesaron matrices cuadradas de 1024, 2048, 4096, 8192 y 16384 filas y columnas.
2. La comparación de los tiempos de ejecución y eficiencia obtenidos por ambas soluciones híbridas.

A continuación se describen las soluciones implementadas. En todos los casos la multiplicación se realiza almacenando las matrices A y C por filas y la B por columnas de manera de poder aprovechar la localidad espacial y temporal de la memoria *cache* en el acceso a los datos.

5.1 Solución Secuencial

Se resuelve secuencialmente el valor de cada posición de la matriz C según la Ecuación 1. La diferencia con la solución secuencial tradicional se basa en que la matriz A es subdividida en filas y la B en columnas de bloques pequeños para aprovechar la localidad de la cache L1.

5.2 Solución híbrida (I)

El algoritmo utiliza una interacción de tipo *master/worker*, donde el *master* trabaja tanto de coordinador como de *worker*. El mismo divide la matriz C en bloques a procesar y posteriormente genera fases de procesamiento. Dado que todos los procesadores tienen la misma potencia de cómputo y que todos los bloques a procesar son del mismo tamaño, todos procesarán (aproximadamente) a la misma velocidad. De esta manera, en cada fase de procesamiento, el *master* reparte las filas de la matriz A y las columnas de la matriz B según el bloque correspondiente a cada *worker*, incluyendo un bloque para él. Procesa su bloque y luego recibe de todos los demás los resultados para poder así pasar a la siguiente fase de procesamiento. La cantidad de fases se calcula de la siguiente manera: si *b* es la cantidad de bloques que se deben procesar y *w* la cantidad de *workers* (incluyendo al *master* que funciona como *worker* también), la cantidad de fases es *b/w*. En esta solución existe un proceso por hoja que internamente genera 8 hilos para hacer su procesamiento.

Es importante tomar en cuenta que cada proceso necesita almacenar las filas de la matriz *A* que va a procesar, las columnas de la matriz *B* y el bloque de la matriz *C* que genera como resultado.

En cada fase, una vez que reparte los bloques a los *workers*, genera los hilos correspondientes para procesar su propio bloque, dividiendo el mismo en filas para que cada hilo procese un subconjunto de las mismas. Los demás procesos *worker* actúan de la misma manera recibiendo datos y enviando sus resultados al *master*.
Se puede resumir el algoritmo de la siguiente manera:

```
Proceso master:
  Divide la matriz C en bloques.
  Para cada fase:
    Comunica las filas correspondientes de la matriz A
    y las columnas correspondientes de la matriz B a
    los procesos worker, según el bloque que va a
    procesar cada uno.
    Genera los hilos y procesa el bloque que le
    corresponde.
```

```
    Recibe los resultados de los procesos worker.
Procesos worker
  Para cada fase:
    Reciben los datos a procesar.
    Generan los hilos y procesan los datos.
    Comunican los resultados al proceso master.
```

La Figura 2 muestra un gráfico representativo de la solución híbrida (I).

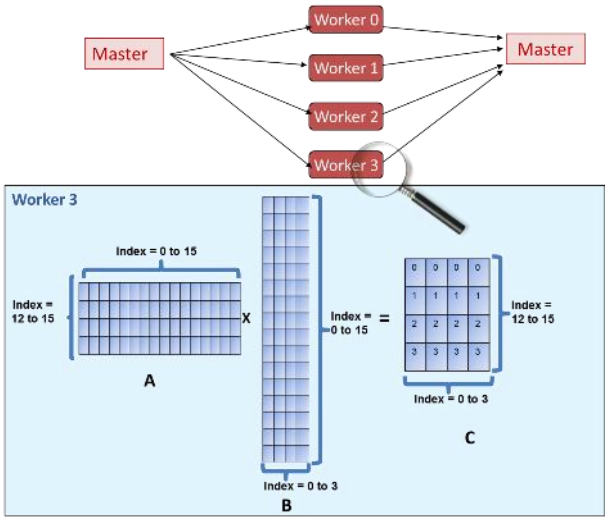

Fig. 2. Solución híbrida (I)

5.3 Solución híbrida (II)

Al igual que en la solución anterior, el algoritmo utiliza una interacción de tipo *master/worker*, donde el *master* trabaja tanto de coordinador como de *worker*. El mismo, divide la matriz A en filas de bloques y la B en columnas de bloques. Dado que todos los procesadores tienen la misma potencia de cómputo y que todos los bloques a procesar son del mismo tamaño, todos procesarán (aproximadamente) a la misma velocidad. De esta manera, el *master* reparte las filas de bloques de la matriz A (según el subconjunto de filas de C que debe calcular cada uno) y todos los bloques de la B a cada *worker*, incluyéndose a sí mismo. Procesa sus filas de bloques de A y luego recibe de todos los demás los resultados. Es importante tener en cuenta que cada proceso necesita almacenar las filas de bloques de la matriz A que va a procesar, todas las columnas de bloques de la matriz *B* y las filas de la matriz *C* que genera como resultado.

Una vez que reparte los datos a los *workers*, se generan los hilos correspondientes para procesar por cada fila de bloques todas las columnas de bloques que la misma posee. Los demás procesos *worker* actúan de la misma manera recibiendo datos y enviando sus resultados al *master*.

Se puede resumir el algoritmo de la siguiente manera:

```
Proceso master:
     1. Divide la matriz A y B en bloques.
     2. Comunica las filas de bloques correspondientes
        de la matriz A y todas las columnas de bloques
        de la matriz B a los procesos worker.
     3. Genera los hilos y procesan la información.
     4. Recibe los resultados de los procesos worker.

Procesos worker
       1. Reciben los datos a procesar
       2. Generan los hilos y procesan los datos.
       3. Comunican los resultados al proceso master.
```

La Figura 3 muestra un gráfico representativo de la solución híbrida (II) con tamaño de bloque 2 *2

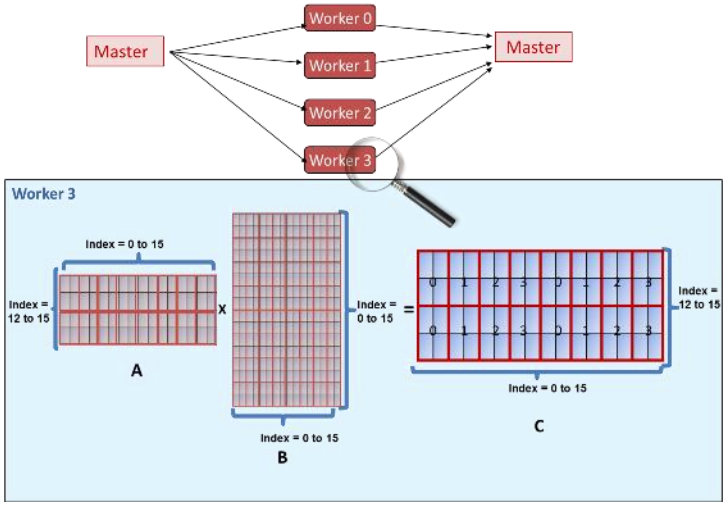

Fig. 3. Solución híbrida (II)

6 Arquitectura utilizada y Resultados Obtenidos

El *hardware* utilizado en la experimentación es un *Blade* de 16 servidores (hojas). Cada hoja posee 2 procesadores *quad core Intel Xeón* e5405 de 2.0 GHz y 10GB de memoria RAM. En todos los casos las características de la misma son las siguientes: memoria *RAM* compartida entre ambos procesadores; *cache* L2 de 2 X 6Mb compartida entre cada par de *cores* por procesador. El sistema operativo utilizado es *Fedora* 12 de 64 bits [17][18].

A continuación se analizan y muestran los resultados obtenidos en las pruebas experimentales realizadas.

En la Tabla 1 se muestran los tiempos de ejecución de la solución secuencial (Sec.). En la Tabla 2, los tiempos obtenidos por la solución híbrida (I) utilizando 16 y 32 núcleos (H(I) 16 y H(I) 32) y el tamaño de los bloques (TBH(I)16 y TBH(II)32) y los obtenidos por la solución híbrida (II) con 16 y 32 núcleos (H(II)16 y H(II)32) y el tamaño de los bloques (TBH(II)16 y TBH(II)32). En todos los casos los tiempos de ejecución están expresados en segundos. En las pruebas se escala tanto la dimensión de la matriz, como la cantidad de núcleos. En la Figura 4 se muestra la eficiencia lograda por ambas soluciones.

Table 1. Tiempos de ejecución secuenciales

Tam. Matriz	Secuencial (seg.)
1024 * 1024	1,62
2048 *2048	12,99
4096 * 4096	103,92
8192 * 8192	831,32
16384 * 16384	6652,68

Table 2. Tiempos de ejecución paralelos

Tam.	TBH(I)16	H(I)16	TBH(I)32	H(I)32	TBH(II)16	H(II)16	TBH(II)32	H(II)32
1024	512	0,24	512	0,23	64	0,25	64	0,25
2048	1024	1,62	1024	1,20	64	1,69	64	1,21
4096	2048	16,85	2048	7,96	64	10,24	64	6,09
8192	1024	127,29	4096	82,90	64	70,12	64	39,22
16384	1024	1212,67	8192	740,87	64	525,44	64	279,97

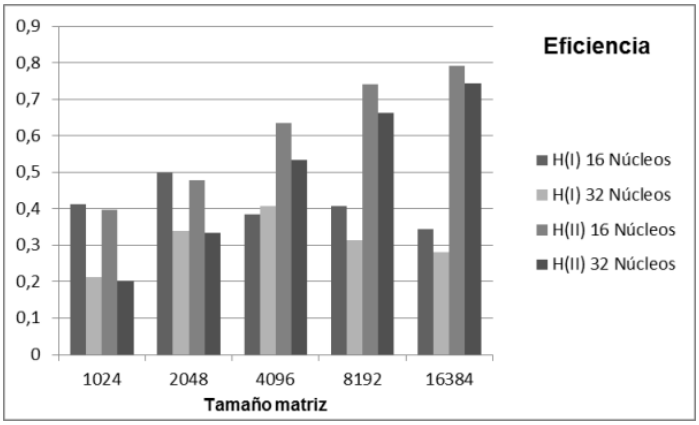

Fig. 4. Eficiencia

En función de los resultados obtenidos, se observa que en la solución híbrida (I) los tiempos de ejecución obtenidos para tamaños de matriz menores que 4096 son en todos los casos mejores que los obtenidos por la solución híbrida (II). Mientras que a partir de 4096, los resultados son exactamente inversos. Esto se debe a que para tamaños menores a 4096, en ambas soluciones los datos necesarios entran

aproximadamente en Cache L1 y la forma de resolución del algoritmo I, al requerir menos operaciones que la solución II, arroja mejores resultados. Mientras que para tamaños mayores, los datos entrarán en Cache L1 para la solución híbrida (II) mientras que no para el algoritmo (I), y el impacto de ello, provoca que los fallos de Cache L1 del algoritmo (I) retrasen notablemente los tiempos de ejecución.

7 Conclusiones y Líneas de Investigación Futuras

Los resultados obtenidos permiten analizar por un lado, que la solución híbrida (II) alcanza una eficiencia mucho mayor al escalar el tamaño del problema del que alcanza el algoritmo (I), ya que se aprovecha la jerarquía de memoria existente en el sistema. Esto se aplica también cuando se escala en la cantidad de núcleos de procesamiento utilizados. Esta diferencia entre un algoritmo y el otro se incrementa al aumentar el tamaño del problema dado que cuando los datos necesarios para ser procesados dejan de entrar en cache L1 en el algoritmo (I), sí lo hacen en el (II), verificando la idea original de la que parte este trabajo que es justamente el impacto del aprovechamiento de la arquitectura en la performance alcanzable por un algoritmo.

Como conclusión podemos analizar que aprovechar la localidad espacial y temporal de los datos en la cache L1, mejora notablemente la eficiencia obtenida a medida que crece el tamaño de los datos (independientemente de la cantidad de núcleos utilizados) a procesar si se lo compara con una solución que no lo aprovecha.

Las líneas de investigación futuras incluyen el análisis de la eficiencia energética de los algoritmos que aprovechan la localidad de los datos y de los que no lo hacen y la adaptación de los algoritmos y su posterior análisis sobre *clusters* heterogéneos [19][20].

8 Referencias

1. Chai L., Gao Q., Panda D. K., “Understanding the impact of multi-core architecture in cluster computing: A case study with Intel Dual-Core System”. IEEE International Symposium on Cluster Computing and the Grid 2007 (CCGRID 2007), pp. 471-478. 2007.
2. Dongarra J. , Foster I., Fox G., Gropp W., Kennedy K., Torzcon L., White A. “Sourcebook of Parallel computing”. Morgan Kaufmann Publishers 2002. ISBN 1558608710 (Capítulo 3).
3. Burger T. “Intel Multi-Core Processors: Quick Reference Guide ”http:// cachewww.intel.com/cd/00/00/23/19/231912_231912.pdf. (2010).
4. http://www.intel.com/support/sp/processors/xeon/sb/cs-007758.htm (2012).
5. Leibovich F., Gallo S., De Giusti L., Chichizola F., Naiouf M., De Giusti A. “Comparación de paradigmas de programación paralela en cluster de multicores: Pasaje de mensajes e híbrido. Un caso de estudio”. Proceedings del XVII Congreso Argentino de Ciencias de la Computación (CACIC 2011). Págs. 241-250. ISBN 978-950-34-0756-1.
6. Leibovich, F., Naiouf, M., De Giusti L., Tinetti, F. G., De Giusti E. “Hybrid Algorithms for Matrix Multiplication on Multicore Clusters”. Julio 2012. WorldComp’12.

7. Andrews G. “Foundations of Multithreaded, Parallel and Distributed Programming”. Addison Wesley Higher Education 2000. ISBN-13: 9780201357523 .
8. Leibovich, F., De Giusti, L., Naiouf, M. “Parallel Algorithms on Clusters of Multicores: Comparing Message Passing vs Hybrid Programming”. Julio 2011. WorldComp’11.
9. Grama A., Gupta A., Karpyis G., Kumar V. “Introduction to Parallel Computing”. Pearson – Addison Wesley 2003. ISBN: 0201648652. Segunda Edición (Capítulo 3).
10. https://computing.llnl.gov/tutorials/openMP (2012).
11. http://www.open-mpi.org (2012).
12. Kumar V., Gupta A., “Analyzing Scalability of Parallel Algorithms and Architectures”. Journal of Parallel and Distributed Computing. Vol 22, nro 1.Pags 60-79. 1994.
13. Leopold C., “Parallel and Distributed Computing. A Survey of Models, Paradigms and Approaches”. Wiley, 2001. ISBN: 0471358312(Capítulos 1, 2 y 3).
14. Chapman B., “The Multicore Programming Challenge, Advanced Parallel Processing Technologies”; 7th International Symposium, (7th APPT'07), Lecture Notes in Computer Science (LNCS), Vol. 4847, p. 3, Springer-Verlag (New York), November 2007.
15. Bischof C., Bucker M., Gibbon P., Joubert G., Lippert T., Mohr B., Peters F. (eds.), Parallel Computing: Architectures, Algorithms and Applications, Advances in Parallel Computing, Vol. 15, IOS Press, February 2008.
16. Becker Alexander (Editor), “Concurrent and Parallel Computing: Theory, Implementation and Applications”, Nova Science Pub Inc, 2008, ISBN-10: 1604562749, ISBN-13: 9781604562743
17. HP, "HP BladeSystem". http://h18004.www1.hp.com/products/blades/components/c-class.html. (2011).
18. HP, "HP BladeSystem c-Class architecture". http://h20000. www2.hp.com/bc/docs/support/SupportManual/c00810839/c00810839.pdf (2011).
19. Feng, W.C., “The importance of being low power in high-performance computing”. Cyberinfrastructure Technology Watch Quarterly (CTWatch Quarterly). 2005.
20. Balladini J., Grosclaude E., Hanzich M., Suppi R., Rexachs D., Luque E., “Incidencia de los modelos de programación paralela y escalado de frecuencia de CPUs en el consumo energético de los sistemas de HPC”. XVI Congreso Argentino de Ciencias de la Computación, pp. 172-181. 2010.